

Introducción al lenguaje MATLAB

1. Descripción del entorno de programación MATLAB.
2. Datos en MATLAB:
 - a) Carácter vectorial de MATLAB.
 - b) Operadores de creación de matrices.
 - c) Tipos de datos usados en MATLAB
3. Programación en MATLAB:
 - a) Entrada / Salida de datos.
 - b) Operadores.
 - c) Sentencias de control de flujo.
4. Modularidad: scripts y funciones.
5. Gráficos en MATLAB:
 - a) Jerarquía de Objetos Gráficos
 - b) Funciones de alto nivel para gráficos

¿Por qué un lenguaje concreto?

- Al describir algoritmos y resolver problemas de computación numérica podríamos quedarnos al nivel de pseudo-código, sin comprometernos en un lenguaje determinado.
- Sin embargo muchas veces el pseudo-código no es más que el lenguaje de programación del profesor/autor, sólo que expresado de una forma más informal, sin ser puntilloso con el léxico.
- Realmente, del pseudo-código a un lenguaje dado no hay un gran salto y es un incentivo para el alumno ser capaz de ejecutar sus algoritmos y ver resultados.
- Tener código “de verdad” y poderlo ejecutar nos permite tratar aspectos como la depuración y optimización de programas.
- Por las razones anteriores, vamos a complementar la teoría y algoritmos presentados en este curso con unas prácticas y clases de laboratorio usando un lenguaje concreto: MATLAB

¿Por qué MATLAB?

- Interfaz de usuario “amigable” que facilita su uso por primera vez.
- Consta de un entorno muy completo con diversas aplicaciones integradas (editor, ventana de comandos, historial de los comandos tecleados, visor de variables en uso, etc.)
- Ampliamente usado en entornos académicos y profesionales.
- Es básicamente un lenguaje interpretado, lo que nos proporciona un inmediato “feedback”. Tras cada orden podemos ver de forma inmediata sus resultados (o los errores cometidos).
- Librerías de funciones muy completas: al poco de familiarizarnos con el podemos estar haciendo cálculos muy complejos.
- Librerías Gráficas incorporadas, facilitando presentación de datos con multitud de formatos gráficos.

Algunas referencias de MATLAB en la WEB

- Página de MATLAB (MathWorks):

<http://www.mathworks.com/>

- Curso MATLAB interactivo: <http://www.imc.tue.nl/>

- Enlace a diversos cursos y tutoriales:

http://www.mathworks.com/academia/student_center/tutorials/launchpad.html

- Aprenda MATLAB como si estuviese en primero:

<http://mat21.etsii.upm.es/ayudainf/aprendainf/Matlab70/matlab70primero.pdf>


ENTORNO de MATLAB

- Como sucede con la mayoría de los lenguajes actuales, las versiones actuales de MATLAB son mucho más que un compilador, comprendiendo diversas aplicaciones tales como:

1. La ventana de ordenes ([command window](#))
2. Espacio de trabajo ([workspace](#))
3. Historial de comandos ([history window](#))
4. Editor ([edit](#))
5. Escritorio de ayuda ([helpdesk](#))

- Para empezar a trabajar, debemos entrar en nuestra primera sesión de MATLAB.

Sesión de Matlab

- Al lanzar MATLAB (icono ) nos aparece un entorno con varias aplicaciones en una única ventana.
- Estas aplicaciones del entorno pueden independizarse (undock) creándose ventanas independientes que pueden ocultarse por separado.
- Al iniciar la sesión cambiaremos el directorio de trabajo (donde se guardan por defecto los programa/resultados de la sesión). También puede modificarse el PATH, que incluye los directorios donde matlab busca datos o programas.
- La ventana principal es la de ORDENES, donde introducimos ordenes tras el cursor (>>)
- Ventanas adicionales pueden abrirse como consecuencia nuestros comandos (hacer un plot provoca la creación de una nueva ventana), o al lanzar nuevas aplicaciones desde los menús.
- Al final de la sesión salimos con **exit**

Ventana de Ordenes (Command Window)

- Es la ventana principal (puede ser la única que usemos)
- En ella se introducen diversos tipos de ordenes:
 1. Sentencias del lenguaje Matlab, (`x=[0:0.1:p]`; `y=sin(x)`; `plot(x,y)`;) que se ejecutan al pulsar enter. Así podemos probar el efecto (o los errores) de cada orden de nuestro futuro programa.
 2. Otras ordenes (`helpdesk`, `edit`, `workspace`) lanzan otras aplicaciones del entorno o nos permiten cambiar directorio de trabajo o el path (`cd`, `path`). Todo esto se puede también hacer desde los menús.
 3. Comandos precedidos por ! (p.e. `!dir`, `!format c`;) no son procesados por MATLAB: se envían directamente al sistema operativo.
- Obviamente de cara a la programación los más importantes son los primeros, que permiten usar la ventana de comandos como un lenguaje interpretado línea a línea, lo que lo hace muy interactivo.

Primeros comandos en MATLAB

- Los primeros comandos introducirán la entrada/salida básica.
- El comando más sencillo de entrada de datos es la asignación (=)

```
>> a=0.5;
```

```
>> a=a+1;
```

```
>> b=1; c=a+b;
```

- La asignación supone que el contenido de la derecha (que debe ser evaluable) se asigna a la variable de la izquierda.
- Obviamente, pese a usar el mismo símbolo (=) no supone una identidad matemática (si no, el comando `a=a+1` no tendría sentido).
- Si se suprime el punto y coma (;) al final de una asignación, el valor final de la variable en cuestión es volcada a pantalla. Con esto tenemos nuestra salida más sencilla.

Espacio de trabajo (workspace)

- El workspace de MATLAB es como se denomina al conjunto de las variables visibles en un momento dado.
- La aplicación `workspace` es una ventana que muestra el nombre, tipo de datos (entero, byte, etc.) y tamaño de las variables presentes.
- Tras los comandos anteriores aparecerían las variables `a`, `b` y `c` en nuestro workspace.
- Desde la línea de comandos, la orden `whos` nos ofrece la misma info.
- Desde dicha aplicación podemos ver sus valores, editarlos, eliminar variables o crear copias. También se pueden representar gráficamente en 2D o 3D.
- Por supuesto, todo lo anterior puede hacerse también con ordenes sencillas, bien desde la línea de comandos, o dentro de un programa.

Guardar/Restaurar el workspace

- Si en un momento de nuestra sesión debemos cerrar matlab, pero tenemos datos que no queremos perder, para seguir trabajando con ellos en otro momento, debemos almacenar el workspace en disco.
- El comando `save fichero` salva todo el workspace (todas las variables actuales) en un fichero llamado `fichero.mat` (la extensión `.mat` es la usada por matlab para guardar datos).
- En la siguiente sesión un `load fichero` nos devolverá a la situación en la que lo dejamos.
- Si sólo queremos guardar algunas de las variables, usaremos la orden `save nombre_fichero x y z`. Al hacer un load aparecerán las variables `x`, `y`, `z` en nuestro workspace con los valores anteriores.
- El formato `.mat` es sólo propio de matlab, por lo que no se usará si queremos exportar nuestros datos (p.e. Imágenes, audio, etc.) para uso de otras aplicaciones.

Historial de comandos (Command history)

- Los comandos introducidos en la ventana principal se almacenan en un buffer y se pueden consultar en el historial de ordenes.
- Evita que la facilidad de MATLAB, que nos permite probar ordenes sin llegar a escribir un programa, no se convierta en un desastre si cuando por fin conseguimos lo que queríamos, no nos acordamos de cómo llegamos a ese resultado.
- De dicha ventana podemos seleccionar un grupo de ordenes y con click en el botón derecho del ratón elegir entre:
 - Volver a ejecutar las ordenes.
 - Copiar, para luego pegar en la ventana de comandos o en un programa.
 - Crear un programa (m-file) a partir de dichas ordenes.
 - Crear un atajo (shortcut) en el entorno Matlab que nos permita ejecutar todas esas ordenes con un simple click de ratón.

Editor de Matlab

- Se invoca con `edit` desde la línea de comandos y es un simple editor de texto que nos permite crear/modificar programas.
- Los programas MATLAB llevan la extensión `.m` y se llaman m-files.
- Típicamente, tras verificar que una serie de ordenes funciona correctamente, las guardaremos en un fichero. Obviamente también podemos crear directamente dicho fichero (sin probar).
- Cuando desde la línea de comandos tecleemos el nombre de dicho fichero, sus contenidos se ejecutarán de forma secuencial. Esto es, habremos creado nuestro primer programa en MATLAB.
- Además de comandos en el editor podemos añadir comentarios. Cualquier texto tecleado después del símbolo (%) será ignorado.
- El editor está especialmente integrado con MATLAB, utilizando un código de colores para distinguir entre comentarios y texto, palabras claves y variables, etc.

Ayuda en MATLAB

- La ayuda en matlab es muy, muy completa.
- La principal aplicación es [helpdesk](#), que nos abre una ventana de ayuda con las típicas opciones de un índice, búsqueda de palabras claves, y una tabla de contenidos.
- Hay también documentación en pdf sobre diversos temas: programación, gráficos, entorno de matlab, compilador, etc.
- Otro aspecto importante de la ayuda es la existencia de programas de demostración. Pueden invocarse desde [helpdesk](#) o directamente desde la línea de comandos con [demos](#).
- Dichos programas, organizados en temas permiten obtener una idea clara de las potencialidades de matlab en las diferentes áreas.
- Consultando el código fuente de las demos, podemos aprender como funcionan.

Organización de datos en MATLAB

- Cualquiera que sea el tipo de datos a usar (enteros, reales, caracteres, o tipos más complejos como estructuras o punteros), MATLAB siempre los almacena en matrices o tablas (arrays).
- Para MATLAB un número escalar es una matriz 1×1 , un vector fila es una matriz $1 \times N$, un vector columna, una matriz $N \times 1$, una imagen en blanco y negro, una matriz $N \times M$, una imagen RGB una matriz de $N \times M \times 3$ (alto, ancho y los tres canales de color), etc.
- MATLAB deriva de MATrix LABoratory, indicándonos que en su origen MATLAB era fundamentalmente una herramienta de Álgebra Lineal numérica (siendo muy potente en ese campo). Luego se le han añadido muchas otras funcionalidades, pero la organización de datos sigue ligada a su pasado matricial.
- Veremos a continuación los operadores básicos para crear, modificar y acceder a matrices. En los ejemplos trabajaremos con números, pero los operadores nos serán válidos independientemente del tipo de datos que usemos.

Creación de matrices en MATLAB

- A estas alturas debemos pensar en una matriz como un contenedor de datos de varias dimensiones, a cuyos elementos accederemos a través de subíndices.
- El operador fundamental para crear una matriz es []

```
>> A=[1 2 3 ; 4 5 6; 7 8 9; 10 11 12];    % Matriz 4x3
>> B=['a' 'b' 'c' 'd']                    % Matriz 1x4
>> C=[1; 2; 3; 4; 5];                     % Matriz 5x1
>> D=7;                                    % Matriz 1x1
>> Q=[];                                   % Matriz vacia
```

- El símbolo ; dentro de [] indica un salto de fila.
- Para crear vectores columna también podemos usar el operador de transponer matrices ' :

```
>> C=[1; 2; 3; 4; 5];                     % Matriz 5x1
>> C=[1 2 3 4 5]';                         % Idem
```

Funciones especiales para crear matrices

- Para ciertas matrices típicas existen funciones especiales:

```
>> A = ones(2,3)      % Matriz 2x3 llena de unos
>> B = zeros(4,2);    % Matriz 4x2 llena de ceros
>> C = zeros(5);      % Matriz 5x5 con ceros
>> D = rand(2,3);     % Matriz de aleatorios (uniforme [0 1])
>> E = randn(N,M);    % Matriz de aleatorios (distribución normal)
>> F = eye(5);        % Matriz identidad tamaño 5x5
>> G = magic(4);      % Crea un cuadrado mágico 4x4
>> v=[1 2 3 4]; H=diag(v); % Matriz diagonal a partir de vector
```

- También son útiles las funciones [size](#), [length](#), [ndims](#), [numel](#) para saber número de dimensiones, tamaño, número de elementos, etc:

```
>> X = rand(1,5);
>> size(X), ans = 1 5 >> size(X'), ans = 5 1 >> length(X), ans = 5
>> A=ones(2,3);
>> ndims(A), ans=2, >> numel(A), ans=6
>> size(A), ans = 2 3 >> size(A,1), ans = 2, >> size(A,2), ans = 3
```


El operador : para la creación de tablas

- El operador : permite crear vectores con datos equiespaciados.

```
>> x=[0:6];           % Equivalente a x=[0 1 2 3 4 5 6];  
>> x=[0:2:6];        % Equivalente a x=[0 2 4 6];  
>> x=['a':2:'k'];    % x=['a' 'c' 'e' 'g' 'i' 'k'];
```

- La sintaxis sería $x = [\text{inicio} : \text{incremento} : \text{final}]$. Si no se especifica incremento, se entiende que es la unidad.
- El incremento no está limitado a valores enteros:

```
>> x=[0:0.1:pi];      % Crea vector desde 0 a pi con salto 0.1  
>> y=sin(x);         % Calcula seno de todo el vector anterior  
>> plot(x,y)         % Muestra resultados de forma gráfica
```

- Como se observa la mayoría de las funciones de MATLAB pueden trabajar de forma vectorial: si reciben un vector o matriz aplican la función a todos los elementos de la matriz, ahorrándonos bucles.

Concatenación de matrices

- El operador [], además de crear matrices puede usarse para concatenar matrices ya existentes, generando matrices mayores.
- La única restricción es que las dimensiones deben ser compatibles

```
>> A = ones(2,4); B=zeros(2,4); C=ones(2,2);  
>> D = [A B]; % Matriz resultante es 2 x 8  
>> D = [A; B]; % Matriz resultante es 4 x 4  
>> D = [A C]; % Matriz 2 x 6  
>> D = [A; C]; % Error, dimensiones no casan  
>> a='hola '; b = 'que tal'; c=[a b]  
c = hola que tal
```

- En el último ejemplo se ve que para MATLAB una cadena de texto es como una matriz (vector) de caracteres, pudiéndose concatenar.
- Una alternativa al uso de [] para concatenar matrices es usar las funciones [cat](#), [vertcat](#), [horzcat](#) o [repmat](#) (replicar una matriz)

Indexado de matrices o tablas

- Para acceder a los elementos individuales de una matriz se usa una notación similar a la matemática con subíndices:

```
>> A=[1 2 3 ; 4 5 6; 7 8 9; 10 11 12];    % Matriz 4x3
>> A(1,2)    % Primera fila, segunda columna (2)
>> A(3,3)    % Tercera fila, tercera columna (9)
```

- La notación de MATLAB es muy potente y permite acceder a varios elementos simultáneamente, utilizando la sintaxis A([lista índices]):

```
>> A=[1 2 3; 4 5 6; 7 8 9; 10 11 12];    % Matriz 4x3
>> A(1,[1 2])    % Primera fila, elementos 1 y 2
>> A([1 3],3)    % Tercer elemento (columna) de filas 1 y 3
>> A(1,:)    % Todos (:) los elementos de la fila 1
>> A(:,2)    % Todos (:) los elementos de la columna 2
>> A([1:3],[1:2])% Submatriz 3x2 de la matriz original
>> A([2 4],[1 3])% Elementos 1 y 3 de las filas 2 y 4
>> A(2:end,3)    % Elementos desde 2 al último de columna 3
```

Reserva previa de memoria

- MATLAB no requiere una reserva previa de memoria pero si se conoce el tamaño final de una matriz es mejor crearla previamente y luego “llenar “ sus casillas en vez de ir añadiendo datos (y hacer crecer la matriz) sobre la marcha.
- La razón es que cada vez que concatenamos datos MATLAB debe reservar memoria para la nueva matriz resultante (más grande). Si esto se repite mucho es muy costoso computacionalmente:

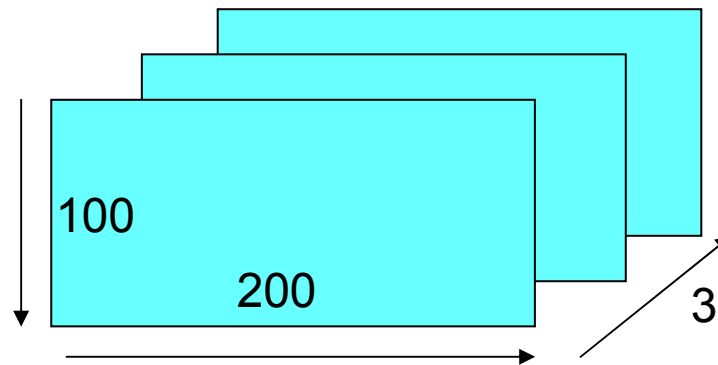
```
>> a=[]; for k=1:10000, a=[a k]; end % Crea vector de 1 a 10000  
>> a=zeros(1,10000); for k=1:10000, a(k)=k; end  
>> a=[1:10000];
```

- Las tres líneas anteriores dan el mismo resultado:
 - a) Se rehace el vector `a[]` a cada paso del bucle, aumentandolo.
 - b) Usa también un bucle, pero reserva previamente la memoria.
 - c) En la última se hace vectorialmente con el comando `[]`
- Hay diferencias significativas de tiempo (medidas con `tic`, `toc`).

Matrices multidimensionales

- En MATLAB no estamos limitados a matrices 2D, pudiendo usar más dimensiones, que se indexarán con 3,4,... índices.

```
>> im=zeros(100,200,3);  
% Tabla 3D de 100 x 200 x 3
```



- La matriz anterior podría ser el contenedor de una imagen RGB, de alto 100 píxeles, ancho 200, y los tres canales (R, G, B) de color.
- El indexado es similar al que hemos visto anteriormente:

```
>> a=im(1:50,1:50,:);           % Subimagen 50x50  
>> a=im(:, :, 2);               % Componente verde(2) de TODA la imagen  
>> a=im(1:2:end,1:2:end,:);     % imagen reducida en factor 2  
>> a=im(:, :, [2 3 1]);         % Imagen con los planos de color permutados
```

Otras peculiaridades en el manejo de matrices

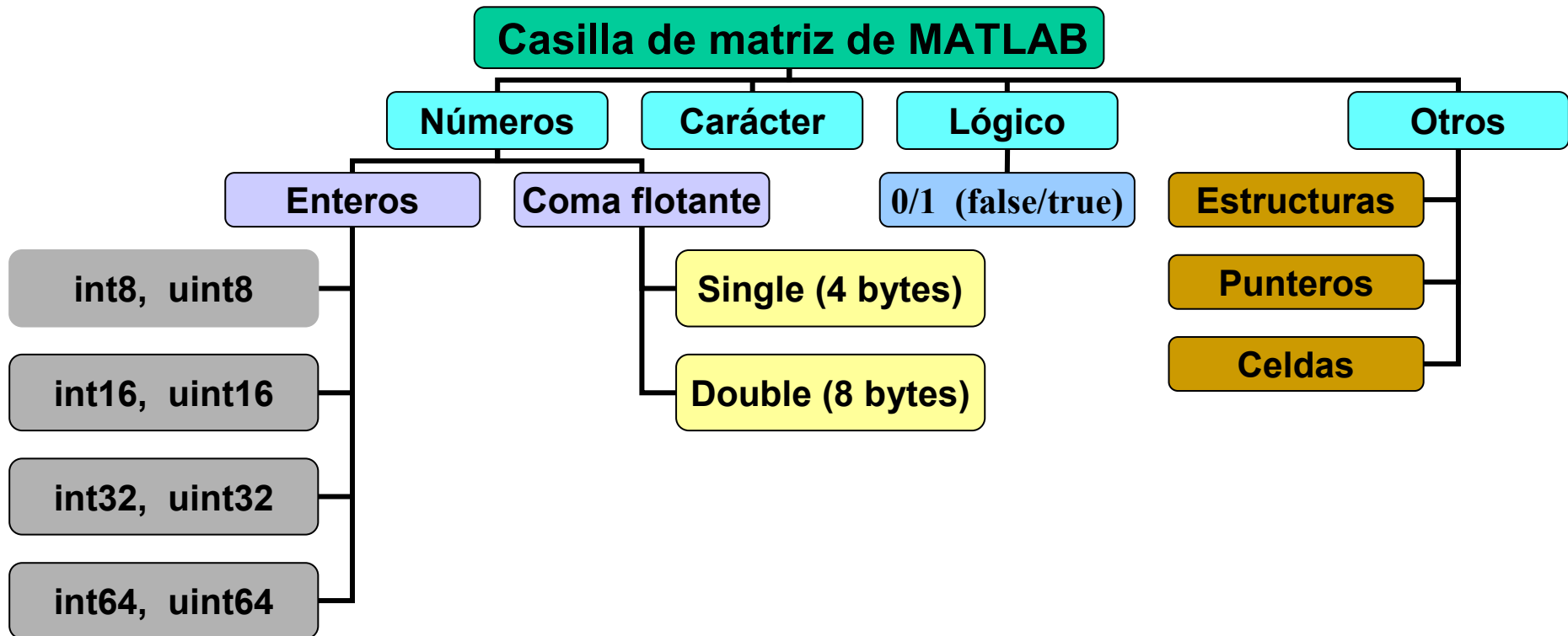
- Los comandos de MATLAB pensados para operar sobre un vector (p.e. mean que calcula la media o un plot para dibujar) al aplicarse a una matriz, operan sobre cada columna por separado.
- Si se desea calcular la media de todos los elementos de una matriz se debe usar A(:), que reordena una matriz en un array 1D:

```
>> A = randn(5);    % Matriz 5x5 de aleatorios en distribución normal.  
>> m=mean(A)        % Media de cada columna (salen 5 valores)  
m =    -0.2176    -0.7995     0.0220    -0.3705     0.1484  
>> m=mean(A(:))     % Media de la matriz entera (único valor)  
m = -0.2434
```

- Una revisión exhaustiva de matrices y tablas en MATLAB está en el [manual de programación en Matlab \(pdf\)](#) dentro del capítulo 1.

Tipos de datos en MATLAB

- ¿Qué tipos de datos podemos guardar en las casillas de un array?
- En un principio MATLAB sólo trabajaba con reales (double), pero actualmente se ha ampliado a otros muchos otros tipos.



Tipos de datos (II)

- **Números:** pueden ser enteros (con distintos rangos, dependiendo del tamaño que reservemos en memoria) o números reales de dos tipos: precisión simple (single) o doble precisión (double).
- **Caracteres:** de texto, representados por su código ASCII o UNICODE, que pueden agruparse formando cadenas de texto.
- **Booleano:** o lógicos. Pueden tomar sólo dos valores: verdadero (1) o falso (0). Son el resultado de comparaciones y operadores lógicos.
- **Estructuras:** similares a las estructuras de C. Una estructura comprende diferentes campos (fields) que pueden guardar diferentes tipos de datos. Los campos se etiquetan para un sencillo acceso.
- MATLAB comprende tipos de datos más complejos que no estudiaremos, como por ejemplo **punteros a funciones** (function handles), o gráficos, celdas (**cells**), clases de Java, etc.
- El usuario puede también definir sus propias clases de datos.

Números enteros

- Hay varios tipos de datos enteros, dependiendo del espacio reservado en memoria, lo que determina su posible rango.

Tipo	Tamaño	Signo	Rango
uint8	1 byte	+	$[0, 2^8)$ [0-255]
int8	1 byte	-/+	$[-(2^7), 2^7)$ [-128,127]
uint16	2 bytes	+	$[0, 2^{16})$ [0-65535]
int16	2 bytes	-/+	$[-(2^{15}), 2^{15})$ [-32768,32767]
uint32	4 bytes	+	$[0, 2^{32})$
int32	4 bytes	-/+	$[-(2^{31}), 2^{31})$
uint64	8 bytes	+	$[0, 2^{64})$

- La elección de un tipo u otro de entero depende del rango de los datos que vamos a manejar (por ejemplo si no van a ser negativos, usaremos uno de los tipos uint, que nos da el doble de rango). Las funciones `intmax()`, `intmin()` nos dan los rangos de un tipo entero

```
>> intmax('int8'), intmax('uint32'),  
ans = 127          ans= 4294967295
```

Números reales (coma flotante)

- Los números reales se guardan en coma flotante (notación científica), como un signo, una mantisa y un exponente (base 2).

Tipo	Tamaño	Mant.	Exp.	Signo	Precisión	Rango
single	4 bytes (32 bits)	23 bits	8 bits	1 bit	6/7 cifras	10^{38}
double	8 bytes (64 bits)	52 bits	11 bits	1 bit	15/16 cifras	10^{300}

- El tamaño dedicado a la mantisa determina la precisión. Para cada cifra decimal precisaremos $\log_2(10)$ (algo más de 3) dígitos binarios.
- El tamaño del exponente determina el rango máximo (2^{exp}).
- Es importante entender que toda operación con números en coma flotante (incluso su simple asignación) conlleva un error, ya que no todos los números reales pueden expresarse con números máquina. La función **eps(x)** nos da la separación entre números máquina para los valores alrededor de x .

Conversión enteros / reales

- Los double son el tipo de datos “original” de MATLAB, por lo que al crearse una tabla o matriz por defecto se crea de tipo double.
- Por la misma razón, la gran mayoría de las funciones de MATLAB trabajan sobre datos de tipo double. Por el contrario cuando usamos enteros nos encontramos que muchas funciones nos dan error:

```
>> x=int8(50); y=sin(x)
```

```
??? Function 'sin' is not defined for values of class 'int8'.
```

- En esos casos debemos hacer una conversión a double antes de aplicar la función correspondiente.
- El problema también se plantea al concatenar datos de tipo entero con double (o con otro tipo de enteros). Una matriz de MATLAB solo puede contener datos de un solo tipo así que o bien MATLAB se queja (error) o los convierte a todos a un tipo común sin decir nada.
- Una recomendación es no dejar que MATLAB haga las cosas por su cuenta: es preferible reconvertir los datos por nuestra cuenta.

Tipo booleano (lógico)

- Sólo pueden tomar dos valores true (1) o false (0), aunque en las asignaciones también se puede usar true o false, para una mejor lectura. Internamente se trabaja con 1/0.

```
>> a = true, b = false,  
a = 1  
b = 0
```

- Aparecen en el resultado de comparaciones (>, >=, <, <=, ==, ~=) o de operaciones lógicas (AND, OR, etc.)
- Se usan como condiciones para controlar el flujo del programa (en los comandos if then, o while) o para el **indexado lógico** dentro de una matriz o vector.

```
>> A=randn(1,1000);  
>> positivos = (A>=0);      % Vector con valores de tipo lógico  
>> B=A(positivos);          % Extracción de valores positivos.
```

Indexado lógico de tablas y matrices

- MATLAB permite indexar una matriz o vector en función de un resultado lógico, lo que permite asignaciones muy compactas.
- Si hacemos una comparación (p.e. mayor que) entre una matriz de MATLAB y otra obtenemos una matriz de elementos lógicos (1 / 0) que nos indica si la condición se cumple (1) o no (0).
- Dicha matriz lógica puede usarse como índice para extraer o operar sólo con aquellos elementos que verifiquen la condición dada:

```
>> x = rand(1,1000);           % Crea vector de 1000 aleatorios entre 0 y 1
>> x(x>0.5) = 1;               % Pone a 1 aquellos que superen el valor 0.5
>> x(x<=0.5) = 0;              % Pone a 0 aquellos que quedan por debajo
```

```
>> a = rand(50); b = rand(50); % Crea dos matrices aleatorias 50x50

>> a(a>b) = b(a>b); % Los elementos de a mayores que el correspondiente
% elemento de b (condición a>b) son substituidos por
% el correspondiente elemento de b.
```

Indexado lógico de tablas y matrices

- En el cuadro siguiente podemos apreciar el estilo más compacto que nos proporciona el indexado lógico.

```
>> a(a>b) = b(a>b);      % Usando indexado lógico y uso vectorial

>> for k=1:50, for j=1:50,    % Usando bucles de forma tradicional
    if a(k,j)>b(k,j), a(k,j)=b(k,j); end
end; end;
```

- Además de operar con un subconjunto de elementos de la matriz, el indexado lógico nos permite extraer aquellos elementos que verifican cierta condición:

```
>> x = rand(1,1000);      % Crea vector de 1000 aleatorios entre 0 y 1
>> y = x(x>0.75)          % Nuevo vector con los elementos de x > 0.75
>> length(y)              % Número de elementos por encima de 0.75.
                           % Deberían ser unos 250 (1000/4)

>> find (x>0.75)          % Índices (en x) de los valores > 0.75
```

Caracteres

- Los caracteres se representan dentro del ordenador con un número (entre 0 y 255, un byte), el llamado código ASCII. Actualmente se usa un código UNICODE de 2 bytes, que permite usar otros alfabetos.
- En la asignación de un carácter se usa la comilla simple ‘

```
>> ch = 'A'
```

- Los caracteres se agrupan en cadenas de texto que para MATLAB no son sino un vector de caracteres (que pueden concatenarse, etc.)

```
>> txt1 = 'hola '; txt2 = 'que tal'; txt = [txt1 txt2],  
txt = hola que tal
```

- Existen funciones (ver [isletter](#), [isspace](#), [isstrprop](#)) para preguntar si los caracteres de una cadena son letras, dígitos, mayúsculas,
- Otras funciones permiten cambiar entre mayúsculas/minúsculas ([lower](#) , [upper](#)), comparar cadenas de texto ([strcmp](#), [strcmpi](#)), encontrar cadenas de texto dentro de otras ([strfind](#)), etc.

Programación en Matlab

- Un programa de MATLAB no es más que una sucesión de ordenes (sentencias) guardadas en un fichero con extensión .m
- Al teclear el nombre del fichero (lo que en MATLAB se conoce como un script) todas sus sentencias se ejecutan secuencialmente.
- En esta sección presentaremos los principales elementos que nos permitirán construir nuestros primeros programas en matlab:
 1. Variables (etiquetas que permiten acceder a los datos)
 2. Entrada / salida en la ventana de comandos.
 3. Operadores (aritméticos, lógicos, comparaciones)
 4. Sentencias de control de flujo del programa: condiciones y bucles

Variables

- Las variables en cualquier lenguaje de programación son etiquetas que se asignan a un dato (o tabla de datos) y nos permiten acceder a su contenido, operar con él, modificarlo, etc.
- Al contrario que C o Pascal, MATLAB no precisa una declaración previa de las variables que vamos a usar al principio del programa.
- En cualquier momento podemos inicializar una nueva variable que precisemos simplemente asignándole un valor.
- Si intentamos usar una variable no inicializada, MATLAB da error.
- Las variables creadas dentro de la ventana de comandos pueden observarse en el workspace y están siempre disponibles a menos que se eliminen con el comando `clear`:

```
>> x=2; y=3; z=[1:10]
>> clear z;    % Elimina variable z
>> clear      % Elimina todas las variables
```

Nombres de variables

- Superando nuestra pereza debemos tratar de dar nombres auto-explicativos a nuestras variables.
- Supone una gran ayuda para que alguien pueda seguir y entender nuestro programa. Lo más probable es que ese alguien seáis vosotros mismos tres meses después intentando modificarlo.
- Los nombres de las variables pueden ser de hasta 63 caracteres y usualmente se usan minúsculas.
- Evitaremos dar a las variables el nombre de funciones existentes.
- Existen algunos nombres reservados que debemos evitar:

`i,j` -> reservadas para la unidad imaginaria

`pi` -> valor de pi (3.141592...)

`Inf, NaN` -> casos especiales de double

`ans` -> matlab guarda siempre el último resultado en `ans`

`computer, version` -> variables con info sobre sistema

Entrada de datos

- Hasta ahora hemos usado la asignación ($a=3.75;$) para introducir datos desde la ventana de comandos. Sin embargo, con la asignación solo podemos introducir datos mientras escribimos el programa
- Para introducir datos **durante la ejecución** de un programa se puede usar el comando `input` que nos permite pedir al usuario que introduzca un valor, acompañándolo de una pequeña descripción:

```
>> temp=input('Introduzca la temperatura (Celsius) ');  
Introduzca la temperatura (Celsius) 27
```

- Al llegar a la sentencia MATLAB nos muestra el texto y espera a que introduzcamos una respuesta. Tras presionar Enter el dato introducido es asignado a la variable que hallamos usado.
- Si MATLAB no entiende lo que escribimos vuelve a preguntarnos. Sin embargo MATLAB no verifica que el tipo de dato sea el esperado. Esto es responsabilidad del usuario usando funciones como `isnumeric`, `ischar`, `isletter`, etc.

Salida de datos (fprintf)

- Omitiendo el punto y coma MATLAB vuelca el contenido de una variable a la ventana de comandos. El formato de esta salida, como por ejemplo nº de decimales, etc se cambia con el comando `format`.
- En la práctica un control más preciso de la salida se consigue con el comando `fprintf`, que además nos permite además mezclar valores de variables (de todo tipo) y un texto descriptivo en la salida.
- Los argumentos básicos fprintf consisten en una cadena de texto de formato y la lista de variables a volcar.

```
>> fprintf(formato, var1,var2,var3,...)
```

- La cadena formato se delimita con comillas simple e incluye texto descriptivo e instrucciones (precedidas por %) sobre como formatear cada variable.
- En la salida los valores de las variables (cada una con un formato propio) se intercalan con el texto descriptivo apareciendo en los lugares donde insertamos su descripción de formato.

Formatos básicos de fprintf

Comando	Descripción	Comando	Descripción
\n	Salto de línea	\t	Inserta espacio (tab)
%d, %i	Volcar entero	%c	Volcar un solo carácter
%x, %X	Formato hexadecimal	%s	Cadena de texto
%f	Volcar double	%e, %E	Notación científica

- Una misma variable puede formatearse con diferentes descriptores:

```
>> a=65;  
>> fprintf('INT %d HEX %x CHAR %c FLOAT %f EXP %e\n',a,a,a,a,a)  
INT 65 HEX 41 CHAR A FLOAT 65.000000 EXP 6.500000e+001
```

- El texto entre descriptores % se vuelca sin cambios. Notar que hay tantos descriptores (5) como variables.
- También podríamos haber usado un array [a a a a a] de cinco datos:

```
>> fprintf('INT %d HEX %x CHAR %c FLOAT %f EXP %e\n',a*ones(1,5))
```

Modificadores en los formatos básicos

- Los formatos básicos pueden modificarse fácilmente:

Comando	Descripción
%5d	Entero con 5 cifras (o espacios hasta completar 5 columnas)
%05d	Entero con cinco cifras insertando ceros delante
%7.3f	Double con siete espacios, 3 de ellos reservados a decimales.
%+10.5f	Double en 10 columnas, volcando signo y 5 decimales.
%12.3e	Double en 12 columnas, notación científica, 3 decimales en mantisa.
%04X	Formato hexadecimal en 4 columnas, rellenando con ceros

- Si hay más descriptores (%) que variables los descriptores de sobra no provocan ninguna salida. Si hay más variables que descriptores el comando se repite hasta agotar las variables especificadas:

```
>> fprintf(' %d ',[1 2 3 4 5])
```

```
1 2 3 4 5 >>
```

Ejemplos de uso de fprintf

- El comando fprintf también puede usarse con una matriz como argumento. Es equivalente a listar TODOS los elementos de la matriz recorrida columna por columna.

```
>> x=[1:5]; >> A=[x; log(x)];  
>> fprintf('log(%5.2f)=%7.4f\n',A)  
log( 1.00)= 0.0000  
log( 2.00)= 0.6931  
log( 3.00)= 1.0986  
log( 4.00)= 1.3863  
log( 5.00)= 1.6094
```

Creamos una matriz cuya 1ª fila son las x's y la 2ª los correspondientes logaritmos. Con un único comando creamos una tabla insertando texto explicativo de los valores volcados. La matriz se recorre columna por columna (x, log, x, log, etc.)

```
>> x=[8:10];  
>>>> fprintf('%d txt\n',x)  
8 txt  
9 txt  
10 txt
```

```
>> x=[8:10];  
>>>> fprintf('%2d txt\n',x)  
8 txt  
9 txt  
10 txt
```

```
>> x=[8:10];  
>>>> fprintf('%02d txt\n',x)  
08 txt  
09 txt  
10 txt
```

Operadores

- Una vez asignados o introducidos los datos en variables el siguiente paso en un programa es operar con ellos.
- Un operador actúa sobre uno o más operandos (variables) y genera un resultado que suele ser asignado a una variable distinta (o a alguno de las variables iniciales, actualizándola):

```
>> x = rand(1,10); y = rand(1,10);
```

```
>> z = x+y; % Asignacion a una tercera variable
```

```
>> x = x+y; % Actualización de uno de las variables
```

- Existen tres clases fundamentales de operadores en MATLAB:

Aritméticos: operan sobre números (suma, multiplicación, etc.)

Comparaciones: mayor que, menor que, etc. Sus resultados son variables lógicas.

Lógicos: operan sobre variables lógicas (AND, OR, etc)

Operadores aritméticos

- Operan con números (o arrays) y el resultado son más números.
- Los símbolos son los comúnmente utilizados: +, -, *, /, ^, etc
- El principal aspecto es distinguir entre operadores matriciales (que siguen las reglas del álgebra lineal) y operadores punto a punto (que operan sobre matrices o arrays elemento por elemento).
- Para diferenciar los operadores punto a punto se les añade un punto . delante del operador (por ejemplo * frente a .*)
- Por ejemplo, dadas A, B dos matrices, $C=A*B$ es la matriz producto, con sus elementos definidos por:

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

- Sin embargo, si usamos el correspondiente operador punto a punto (.*), la matriz resultante $C=A.*B$ tiene por elementos:

$$C_{ij} = A_{ij} \times B_{ij}$$

Operadores matriciales / punto a punto

- Cuando se opera con operadores matriciales se deben respetar las reglas adecuadas en las dimensiones de las matrices. Por ejemplo, en la multiplicación matricial $C=A*B$, el número de columnas de A debe ser igual al número de filas de B. La matriz resultante tendrá las filas de A y las columnas de B.
- En cambio si hacemos $C=A.*B$, ambas matrices A y B deben ser de iguales dimensiones. C también será de las mismas dimensiones.
- Por ejemplo, para poder usar la multiplicación $*$ con vectores partimos de un vector fila a ($1 \times n$) y un vector columna b ($n \times 1$). El resultado $c=a*b$ será un escalar (producto escalar de vectores):

$$c = \sum_k a_k b_k$$

- Si intentamos multiplicar ($*$) dos vectores fila MATLAB da un error. Si podemos multiplicarlos usando $.*$ obteniendo un tercer vector fila:

$$c_k = a_k b_k$$

Operadores matriciales / punto a punto

- En algunos casos (+ / -) la suma de matrices coincide con la suma elemento a elemento, por lo que no existen los operadores (.+) ni (-.).

$A + B$	Suma de dos matrices $n \times m$
$A - B$	Resta de matrices $n \times m$
$-A$	Cambio de signo de A
$A * B$	Multiplicación matricial $(n \times k) * (k \times m) \rightarrow n \times m$
$A .* B$	Multiplicación punto a punto $(n \times m) .* (n \times m) \rightarrow n \times m$
A / B	“Division” matricial. En general equivale a $A * \text{inversa}(B)$
$A ./ B$	División punto a punto $A(i,j)/B(i,j)$
$A ^ k$	Potencia k-ésima de A. Equivale a $A * A * \dots * A$ k veces.
$A .^k$	Eleva a la potencia k cada elemento de A , $A(i,j)^k$
$A '$	Traspuesta conjugada de A
$A .'$	Transpuesta de A (sin conjugar)

Operaciones de matrices con escalares

- La única excepción a la regla de usar matrices de iguales (o compatibles) dimensiones es cuando uno de los operandos es un escalar.
- En ese caso se entiende que se hace una operación punto a punto entre cada elemento de la matriz y dicho escalar.
- Si A es una matriz $n \times m$ y c un escalar, $A+c$ es la matriz $n \times m$ que resulta de sumar c a todos los elementos de A . De igual forma las matrices $A*c$ o A/c son matrices $n \times m$ resultado de multiplicar (o dividir) cada elemento de A por el escalar c .
- Esto sólo funciona para escalares. No podemos sumar un vector fila ($1 \times m$) a una matriz $n \times m$ y esperar que MATLAB entienda que queremos sumar el mismo vector a todas las filas de la matriz:

```
>> A=rand(3,5); x=rand(1,5); % Matriz 3 x 5 y vector 1x5 >>
B=A+c, ??? Error using ==> plus Matrix dimensions must agree.

>> C=ones(3,1)*c; % Matriz 3x5 de filas iguales replicas de c >>
B=A+C; % Correcto
```

Operadores de comparación

- MATLAB proporciona los siguientes operadores de comparación:

Operador	Descripción	Operador	Descripción
>	Mayor que	<=	Menor o igual que
>=	Mayor o igual que	==	Igual que
<	Menor que	~=	Distinto de

- El resultado de una comparación es un valor lógico (1 si se cumple o 0 si no se cumple). Si se opera con arrays, ambos deben ser del mismo tamaño y el resultado será un array de 1/0 lógicos resultado de las comparaciones individuales de todos los elementos.
- La excepción a lo anterior es la comparación de una tabla con un escalar, donde se comparan todos los elementos con dicho escalar.

```
>> x=[1:5]; y=[5:-1:1];
```

```
>> (x<=y) ,    ans =    1        1        1        0        0
```

```
>> (x~=3) ,    ans =    1        1        0        1        1
```

Operadores lógicos

- Realizan las típicas operaciones lógicas AND, OR, NOT, etc.
- Hay dos tipos de operadores lógicos en MATLAB:
 - Operadores sobre variables lógicas. Se usan mucho para combinar varias condiciones y determinar si todas se han cumplido (con un AND), o si al menos una se cumple (con un OR), etc

Operación	Símbolo	Descripción
AND	a & b	TRUE sólo si a y b son TRUE
OR	a b	TRUE si a o b son TRUE
NOT	~ a	TRUE si a es FALSE y viceversa.
XOR	xor(a,b)	TRUE sólo si una entrada lo es y la otra no

- Operadores lógicos que actúan sobre los bits individuales (0/1) de números enteros (**bitand**, **bitor**, **bitcmp**, **bitxor**) . Se usan más raramente y no los trataremos.

Tablas resumiendo los operadores lógicos

AND (a&b)

a\b	0	1
0	0	0
1	0	1

OR (a|b)

a\b	0	1
0	0	1
1	1	1

XOR (xor(a,b))

a\b	0	1
0	0	1
1	1	0

- Ejemplos de aplicación en MATLAB:

```
>> x=5; y=2; a=[1 2 3 4 5];
```

```
>> (x>3) & (y>1), ans = 1
```

```
>> (x>3) | (y>3), ans = 1
```

```
>> ~(x>3), ans = 0
```

```
>> (a>2) & (a<5), ans = 0 0 1 1 0
```

```
>> (x>3) & (y>3), ans = 0
```

```
>> (x>5) | (y>3), ans = 0
```

```
>> xor((x>3), (y>1)), ans = 0
```

- La combinación de varias condiciones se usará en las sentencias condicionales, donde un cierto código se ejecuta (o no, usando NOT) código si se cumplen todas (AND) o alguna (OR) de las condiciones.

Precedencia de operadores

- Si una expresión en MATLAB es compleja (muchos operadores) hay que tener en cuenta la precedencia entre dichos operadores. Por ejemplo $5*3/2*5$ se evalúa como $5*(3/2)*5$ y no como $(5*3)/(2*5)$.
- En el help o los manuales de MATLAB podéis encontrar detalles sobre el orden en que MATLAB evalúa una expresión que combine diversos operadores (incluso mezclando operadores aritméticos, lógicos, etc)
- Sin embargo, la recomendación es no complicar las expresiones, evaluando las diferentes partes por separado y combinándolas posteriormente.
- Ante cualquier duda de precedencia, usar liberalmente los paréntesis. Los paréntesis tienen la más alta precedencia en MATLAB y siempre se evalúa su contenido antes de seguir operando.

Sentencias de control de flujo en el programa

- Una parte fundamental de cualquier programa es la posibilidad de actuar de forma distinta ante diferentes condiciones.
- Se entiende por sentencias de control de flujo aquellas que permiten ejecutar cierto código en función de una condición, repiten unas instrucciones muchas veces (bucle), terminan el programa, etc.
- En MATLAB destacaremos:
 1. Saltos condicionales: `if then else` , `switch case`
 2. Bucles: `for`, `while`, `continue`, `break`
 3. Terminación de un programa: `return`

Condiciones (if, if else)

- La sentencia **if** evalúa una expresión lógica y ejecuta bloques de código basándose en su resultado. Debe terminarse con un **end**

```
if (expresión lógica)
    codigo;
end
```

```
if (expresión lógica)
    código 1;
else
    codigo 2;
end
```

- En el primer caso si la expresión es cierta (true ó 1) se ejecuta el código y si no lo es (false ó 0) se salta. En el segundo se ejecuta el código 1 si la expresión es cierta y el código 2 si resulta ser falsa.

```
if rem(a,2)==0,      % Si el resto de a/2 es cero
    disp ('a es par');
else
    disp('a es impar');
end
```

Condiciones con múltiples casos (switch)

- La sentencia **if else end** es especialmente adecuada en una decisión binaria (p.e. $a > 2$?). Si hay que ejecutar acciones distintas en función de que a valga 1, 2, ó 3 el uso de **if else** se hace confuso, siendo preferible el uso de **switchcase, end**:

CÓDIGO IF ELSE END

```
if (opt==1), COD1;
else
    if (opt==2), COD2;
    else
        if (opt==3), COD3;
        else
            OTROS CASOS;
        end
    end
end
end
```

CÓDIGO SWITCH CASE END

```
switch (opt)
    case 1: COD1;
    case 2: COD2;
    case 3: COD3;
    otherwise: OTROS CASOS;
end
```

Se observa la mayor claridad del código de la derecha. A cada alternativa le sigue el código correspondiente. El código detrás de **otherwise** se ejecuta si no se verifican ninguna de las condiciones listadas. Es opcional y si no se pone no se ejecuta ningún código.

La variable **opt** usada en el **switch()** puede ser un escalar o una cadena.

Iteraciones y bucles

- Una de los principales ventajas de un programa es la posibilidad de repetir una tarea un número determinado de veces o bien hasta que se cumpla una cierta condición.
- En MATLAB, para el primer caso se usa la sentencia **for**, que nos permite dar una **lista de valores** a un índice y un **fragmento de código** que se ejecutará para los valores especificados del índice:

```
for k = lista
    CODIGO;
end
```

Típicamente lista es un vector equiespaciado expresado como (inicio : inc : final). k varía desde inicio hasta fin con salto inc. Si se omite el incremento (ini : fin) se usa la unidad.

El ejemplo siguiente aproxima el valor del número e sumando N términos del desarrollo de Taylor de la función e^x desarrollada en $x=1$:

```
>> temp=1; e_aprox=1; N=50;
>> for k=1:N, temp=temp/k; e_aprox = e_aprox + temp; end
```

Bucles FOR

- La lista barrida por k puede ser cualquier vector, no precisándose que sea equiespaciado ni monótono. Por ejemplo podríamos tener for k=[1 -4 8 1], donde k tomaría los valores 1, -4, 8, y 1 de nuevo.
- Podemos anidar bucles, teniendo en cuenta de que para cada valor del índice del bucle más exterior se ejecutarán todos los valores del índice del bucle anidado:

```
for k=1:1000, for n=1:1000,  
    A(k,1) = A(k,1) - 5; % Por aquí se pasa un millón de veces  
end, end
```

- Nuestra lista también puede ser una matriz, en cuyo caso, en cada iteración k toma el valor de las sucesivas columnas de la matriz.
- Conviene tener en cuenta que en muchos casos nuestro programa puede optimizarse eliminando bucles FOR, al aprovechar el carácter vectorial de MATLAB. El bucle anterior puede cambiarse por A=A-5.

Bucles WHILE

- La sentencia WHILE nos permite repetir cierto código hasta que una condición lógica se verifique:

```
while (condicion),  
    CODIGO;  
end
```

Las sentencias comprendidas entre el while y end se repiten mientras la condición se evalúe como cierta.

- Si la condición resulta ser falsa la primera vez el código dentro del bucle no se ejecuta nunca.
- El código siguiente es una mejora en la aproximación al número e:

```
temp=1; e_aprox=1; k=1;  
while (temp>1e-16)  
    temp=temp/k;  
    e_aprox = e_aprox + temp;  
    k=k+1;  
end
```

Saltando pasos y saliendo de un bucle

- La sentencia `continue` nos permite saltar una iteración dentro de un bucle FOR. En cuanto aparece se deja de ejecutar la iteración en la que nos encontramos y se pasa a la siguiente.
- La sentencia `break` (dentro de un bucle FOR o de un WHILE) hace que abandonemos el bucle y no se ejecuten las iteraciones que faltan.

```
for k = 1:10,  
    if k==7, continue; end  
    disp(k^2)  
end
```

```
for k = 1:10,  
    if k==7, break; end  
    disp(k^2)  
end
```

- El código de la izquierda vuelca los cuadrados de los números del 1 al 10, pero se salta el del 7 debido al `continue`.
- El código de la derecha vuelca los cuadrados del 1 al 6, ya que al llevar al `break` no se sigue con el bucle.
- En el caso de bucles anidados un `break` no sale de todos ellos, sólo devuelve el control al bucle inmediatamente por encima.

Terminación prematura de un programa

- Una vez que MATLAB ha terminado de ejecutar las sentencias de un programa devuelve el control a la ventana de comandos.
- Si se desea acabar un programa antes de llegar a su final se usa `return`. Si encontramos un `return` se ignoran el resto de las sentencias. Es el equivalente de un `break` dentro de un bucle.
- Un típico uso consiste en detener la ejecución del programa si se detecta un error (previo mensaje al usuario):

```
.....  
  
if (size(A,2) ~= size(B,1))      % no coinciden dimensiones de matrices  
    disp('Las matrices no pueden multiplicarse');  
    disp('Nos largamos');  
    return  
end  
  
C=A*B;  
.....
```


Programas en MATLAB

- Una vez conocidos los bloques básicos de MATLAB solo queda combinarlos en una estructura superior (programa).
- En su forma más simple un programa de MATLAB es una sucesión de ordenes (sentencias) guardadas en un fichero con extensión .m
- Al teclear el nombre del fichero en la ventana de comandos las sentencias se ejecutan de forma secuencial.
- En principio, cualquier programa, por complicado que sea podría escribirse en un único fichero.
- Sin embargo, una buena práctica de programación aconseja subdividir el programa en diversos módulos
- Cada uno de dichos módulos cumple una función determinada y se debe guardar en ficheros .m separados.
- Hay dos tipos principales de ficheros .m : scripts y funciones.

Scripts

- Un fichero de tipo script es una simple lista de comandos.
- Su efecto es idéntico a insertar los comandos que contiene el script en el punto del programa donde le llamamos:

```
%% Fichero script llamado contraste.m  
t1=(a+b); t2=abs(a-b);  
c=t2/t1;
```

```
a=4;b=5;
```

```
contraste
```

```
disp(c)
```



```
a=4;b=5;
```

```
t1=(a+b); t2=abs(a-b);
```

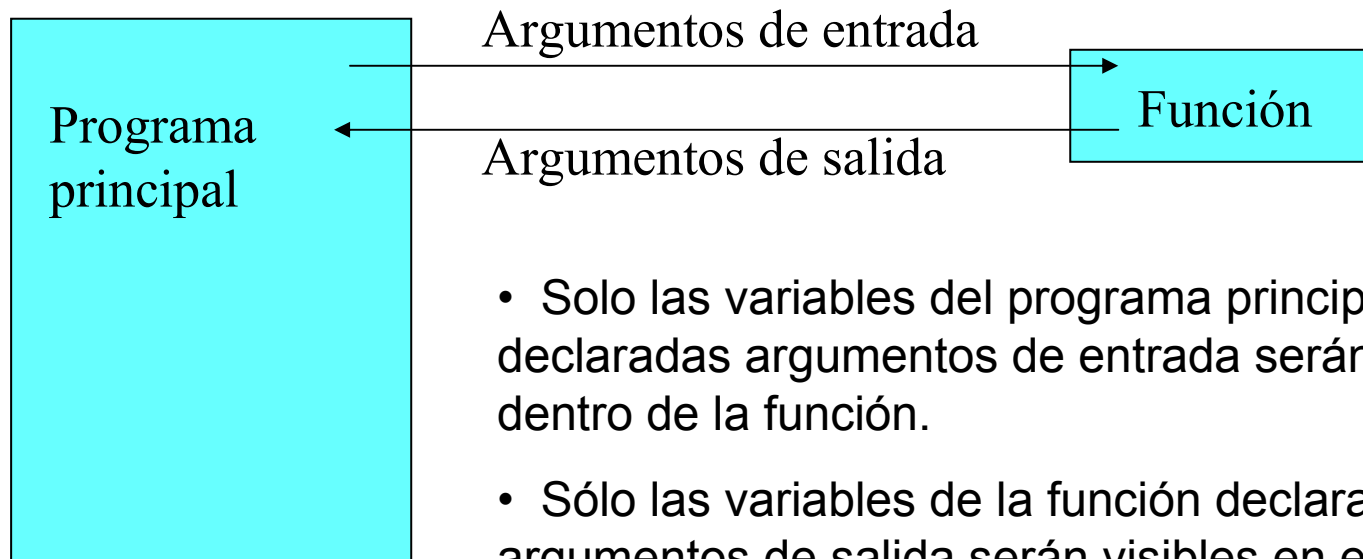
```
c=t2/t1;
```

```
disp(c)
```

- Un script **comparte variables** con el programa que le llama (se ven las que había y las que se crean se añaden al workspace): en este caso precisamos que existan dos variables a y b antes de llamar al script, y, al terminar, contamos con dos variables adicionales (t1 y t2)
- No acepta argumentos de entrada (solo trabajan con a y b) ni devuelve argumentos de salida (el resultado siempre está en c).

Funciones en MATLAB

- Debido a que comparten las variables con el programa que les llama los scripts no son una buena opción para nuestro objetivo de modularizar nuestro programa.
- Las funciones en MATLAB tienen su propio espacio de variables y no ven las variables del programa que las llama. La única forma de relacionarse es a través de los argumentos de entrada y salida.



- Solo las variables del programa principal que sean declaradas argumentos de entrada serán visibles dentro de la función.
- Sólo las variables de la función declaradas como argumentos de salida serán visibles en el programa principal.

Funciones en MATLAB

- Repitamos el ejemplo anterior usando una función en vez de un script

```
%% Fichero .m con una función llamada contraste.m  
  
function c = contraste(a,b)  
  
t1=(a+b); t2=abs(a-b); c=t2/t1;
```

```
% Llamada desde el programa principal
```

```
M=5; m=2; q=contraste(m,M);
```

- a y b representan los argumentos de entrada en la función, pero podemos usar cualquier par de variables en el programa principal: la función hará con ellas lo que haga con a y b en su definición.
- El valor del argumento de salida (c) se asignará a la variable que se especifique en el programa principal (q).
- Las variables usadas internamente t1, t2, c no son visibles desde el programa llamador.

Ventajas del uso de funciones

- Con las funciones si que podemos dividir nuestro programa en módulos “estancos”, que sólo compartan la mínima información necesaria: argumentos de entrada y salida.
- Permite dividir el problema en tareas sencillas con objetivos claros: por ejemplo, recibir datos (x, y) para calcular un cierto resultado z .
- El cómo codificar dicha función no es un problema que nos preocupe al diseñar el programa principal. Puede dejarse para luego o incluso encargárselo a otra persona. No hay peligro de que las variables intermedias usadas interfieran con las que yo uso en el principal.
- Esta es la clave para el desarrollo de programas complejos: subdividirlos en unidades más pequeñas con cometidos más sencillos y con la menor interacción posible entre ellos: unos pocos argumentos de entrada y de salida.
- Las unidades sencillas (funciones) son más fáciles de verificar.

Definición de una función MATLAB

- Veamos en detalle los elementos de un fichero .m que contenga la definición de una función

Indica que es una función MATLAB

```
function f = fact(n)
% Calcula el factorial de n: n!

f=1;
for k=1:n, f=f*k;
```

Argumento/s salida

Nombre función

Argumento/s entrada (entre paréntesis)

Comentarios explicativos sobre uso de la función, argumentos entrada/salida, etc. Aparecen al hacer un `help fact`

Núcleo de la función, donde se resuelve el problema. Durante el proceso debe asignarse valores a los argumentos de salida (en este caso f)

Argumentos de entrada

- Si hay varios argumentos de entrada se separan por comas.
- Usando `nargin` (n arg in) podemos saber con cuantos argumentos de entrada se ha hecho la llamada. Esto da más flexibilidad a las funciones:

```
function x = aleat(N,sigma,m)

% Genera un vector de N aleatorios con desviación sigma y media m
% Si no se especifican sigma=1 y m=0

x=randn(1,N);  if nargin==1, return; end

x=x*sigma;     if nargin==2, return; end

x=x+m;

>> a = aleat(100);      % Vector 1x100 con sigma=1 y media 0
>> a = aleat(100,2);    % Vector 1x100 con sigma=2 y media 0
>> a = aleat(100,3,-1) % Vector 1x100 con sigma=3 y media -1
```

- El uso de `nargin` evita realizar trabajo extra si el usuario no lo requiere y permite asignar valores por defecto a los parámetros de entrada.

Argumentos de salida

- Una función puede devolver varios resultados: se especifican como argumentos de salida entre corchetes y separados por comas.
- Igualmente, usando `nargout` (n arg out) podemos saber cuantos argumentos de salida nos ha requerido el usuario:

```
function [m,s]= stat(x)

% Calcula la media m y la varianza s de un vector de datos x[]
m=mean(x);

if nargout==2, s=std(x); end
```

```
>> m = stat(x);      % Calcula la media de x
>> [m s] = stat(x); % Calcula la media Y la desviación estándar de x
```

- De nuevo, el uso de `nargout` evita que se realice trabajo adicional si el usuario no lo requiere.

Variables locales, globales y persistentes

- En principio las variables solo son visibles dentro de su workspace, y cada función (y el programa principal) tienen distintos workspaces. Es por eso por lo que es necesario el uso de argumentos (de entrada y de salida) para comunicarse entre sí.
- Por otra parte el workspace de una función se crea al llamarla y se borra al terminar, de forma que sus variables no se conservan entre llamadas a la función.
- Esta es la situación por defecto: se dice que las variables son **locales**.
- Si se desea alterar esta situación hay que declarar las variables de forma especial como **globales** o **persistentes**.
- Una variable que se declare **global** en varias funciones (o en el programa principal) puede verse desde todas esas funciones.
- Igualmente, una variable que se declare **persistente** en una función mantiene su valor entre sucesivas llamadas a la misma función.

Uso de variables globales y persistentes

- Las variables **globales** se usan cuando hay muchos parámetros que deben ser usados por varias funciones y no resulta cómodo pasarlos todos como argumentos.
- Utilidad de una variable **persistente**: contador que lleve la cuenta de las veces a las que se llama una función (por ejemplo para evaluar el coste computacional de un algoritmo)
- En general el uso de variables no locales no es aconsejado (viola el principio de dividir el programa en unidades estancas) y debe usarse sólo si no vemos una buena alternativa.
- Es una buena práctica usar las mayúsculas para variables no locales.

Ejemplo de variables globales y persistentes

```
function uno()  
global M  
M=3; fprintf('M=%d\n',M);
```

```
function dos()  
global M  
M=M+1; fprintf('M=%d\n',M);
```

```
>> uno  
M = 3  
  
>> dos  
M = 4
```

Desde la segunda función se ve el valor asignado a M en la primera, dado su carácter global

```
function acumula(x)  
persistent suma_x  
if isempty(suma_x) suma_x=0; end  
suma_x=suma_x+x;
```

Al declarar una variable persistente por primera vez se crea como una variable vacía []. Esta sentencia comprueba si suma_x está vacía y caso de ser así la inicializa a cero antes de usarla por primera vez.

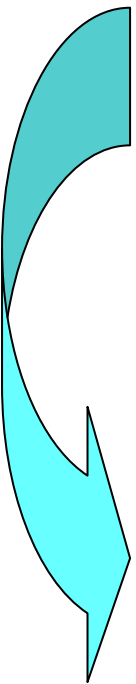
```
>> acumula(2)  
suma_x = 2  
  
>> acumula(3)  
suma_x = 5
```

El valor de suma_x se mantiene (persiste) entre llamadas a la función acumula().

Subfunciones

- Si una función es suficientemente complicada puede ser aconsejable subdividirla en otras funciones. En ese caso tenemos dos alternativas:
 1. Escribir las nuevas funciones en ficheros .m independientes, como funciones “de pleno derecho”. Dichas funciones podrán ser llamadas desde cualquier otro programa o función.
 2. Escribirlas en el mismo fichero .m, a continuación de la función original. En este caso reciben el nombre de subfunciones y sólo son visibles desde la función principal (la primera del fichero .m). Sólo la función principal es visible para el resto del programa.
- Usaremos el segundo enfoque si las funciones auxiliares sólo resultan de interés para la función original (no van a ser llamadas desde otras funciones). Así están organizadas y se evita la proliferación de ficheros m, cada uno con una única función.
- Obviamente, si deseamos usarlas desde otras funciones (o el programa principal) debemos guardarlas en su propio fichero.

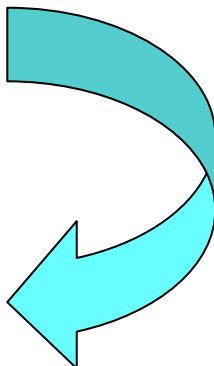
Uso de subfunciones



```
function [m,s] = stat(x)
m=media(x);
s=varianza(x);
return

function m=media(x)
M=sum(x)/length(x);
return

function var = varianza(x)
...
...
```



Todo esto dentro
de un fichero
llamado stat.m

Sólo stat() puede llamarse desde el programa principal o desde otras funciones. Si intentásemos llamar a media() o a varianza() tendremos un error, indicándonos que MATLAB no es capaz de encontrarlas.

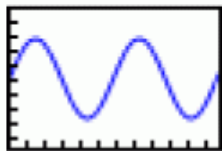
Gráficos en MATLAB

- MATLAB ofrece unas librerías y herramientas muy potentes para la visualización de datos en varias dimensiones.
- Una de las ventajas de MATLAB es que podemos crear gráficos con un mínimo de esfuerzo usando comandos de “alto nivel” como plot o surf.
- Si queremos podemos también utilizar accesos de bajo nivel que permiten leer/modificar cualquier característica de un gráfico usando las funciones get/set aplicadas al correspondiente puntero (o handle) del objeto gráfico correspondiente
- Para poder aprovechar al máximo las posibilidades de MATLAB hay que tener una idea básica del enfoque orientado a objetos que usa MATLAB en sus aplicaciones gráficas, así como la jerarquía entre los distintos objetos gráficos que existen.

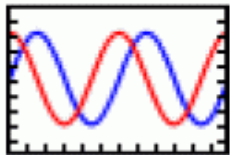
Ejemplos de gráficos en MATLAB (2D)

Line Graphs

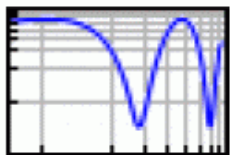
plot



plotyy

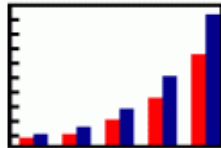


loglog

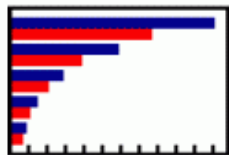


Bar Graphs

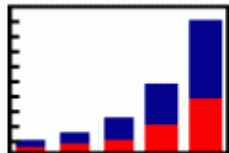
bar
(grouped)



barh
(grouped)

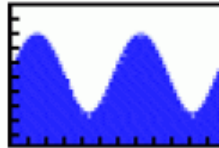


bar
(stacked)



Area Graphs

area



pie

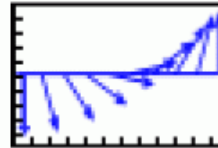


fill

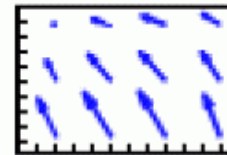


Direction Graphs

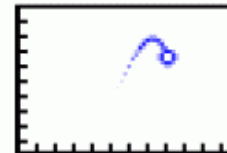
feather



quiver

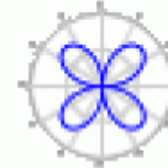


comet



Radial Graphs

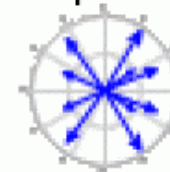
polar



rose

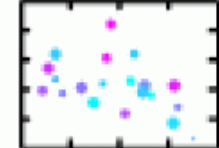


compass

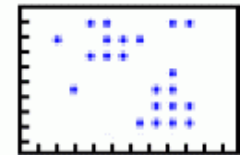


Scatter Graphs

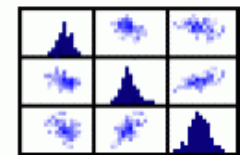
scatter



spy



plotmatrix



Ejemplos de gráficos en MATLAB (3D)

Line Graphs

ezplot3

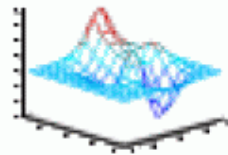


waterfall

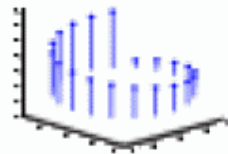


Mesh Graphs and Bar Graphs

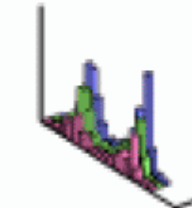
ezmesh



stem3



bar3



Area Graphs and Constructive Objects

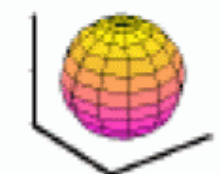
cylinder



ellipsoid

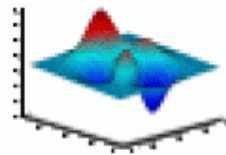


sphere

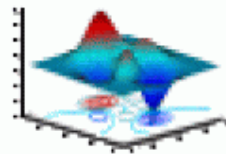


Surface Graphs

ezsurf



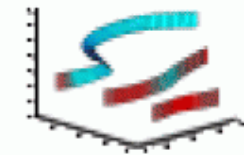
ezsurfz



Direction Graphs

Volumetric Graphs

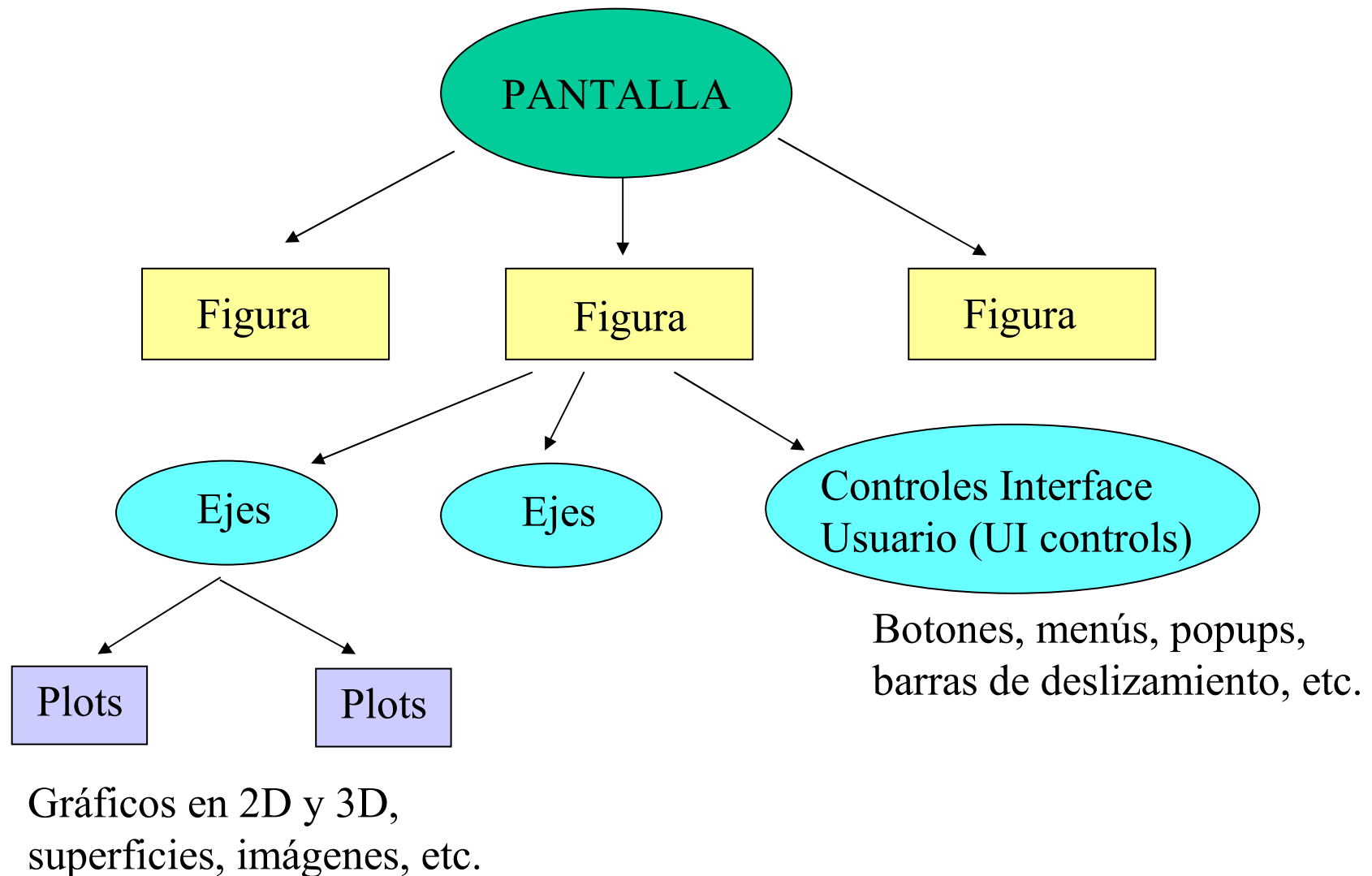
streamribbon



streamtube



Jerarquía de gráficos en MATLAB



Objetos gráficos.

- Cada uno de los niveles del gráfico anterior (figuras, ejes, plots) son objetos independientes cuyas propiedades pueden modificarse individualmente.
- Hay una jerarquía de padres e hijos: un plot sólo puede pintarse en unos ejes que a su vez deben estar en una figura.
- Si eliminamos (cerramos) una figura, eliminamos todos los objetos que contenga. Por el contrario, podemos eliminar unos ejes de una figura sin afectar a otros componentes (otros ejes, controles, etc.)
- No se puede crear un objeto sin los correspondientes ancestros. Esto es, podemos crear una figura por si sola pero no un plot aislado.
- Cuando usamos una función de alto nivel (plot) MATLAB automáticamente crea una figura y unos ejes donde pintar el gráfico.
- Por eso podemos hacer gráficos en MATLAB fácilmente sin tener que ser conscientes de la existencia de diferentes objetos gráficos organizados en una jerarquía.

Objeto gráfico figura

- Las figuras (o ventanas) de MATLAB se crean (vacías) con la función `figure()` que devuelve un puntero a la ventana creada.
- Si se usa sin argumentos se crea una nueva figura. Si se usa un puntero a ventana como argumento, hace activa dicha ventana.
- El comando `gcf` (get current figure) nos dice la ventana activa:

```
>> f1=figure,          f1 = 1
>> f2=figure,          f2 = 2
>> gcf,                ans = 2
>> figure(f1); gcf, ans = 1
```

- La función `close(f1)` cierra la ventana `f1`. `close all` cierra todas
- Las propiedades de una figura (bajo nivel) pueden cambiarse con un `set()`:

```
>> set( f1, 'Position', [10 10 600 400], 'Color', 'r' );
>> set( f1, 'Number', 'off', 'Name', 'Mi ventana', 'Resize', 'off' );
```

Menú de las Figuras

- Las figuras de MATLAB por defecto se crean con un menú.
- Desde el menú se pueden realizar acciones diversas como acceder a las propiedades de los distintos objetos gráficos (View / Property Editor o dentro de Edit) para verlas o modificarlas.
- También se pueden cambiar el ángulo de visión de la figura, condiciones de iluminación, añadir anotaciones, flechas, etc.
- Uno de las acciones más importantes es guardar una figura.
- Una figura puede guardarse en un formato propio de MATLAB con extensión fig. En ese caso al cargar de nuevo la figura está como la dejamos y podemos seguir operando con ella.
- También puede guardarse en un formato gráfico tipo bmp, jpeg o similares (para incluirla en un documento). Hay que tener en cuenta que en ese caso la figura se “colapsa” y se convierte en una imagen y ya no es posible acceder independientemente a los objetos gráficos que la componían (ahora sólo tenemos píxeles).

Objeto gráfico ejes

- Al llamar a un plot, el gráfico se pinta en los ejes activos. Si no existe ningún eje se crean en la figura activa (y si hace falta se crea la figura correspondiente). Dichos ejes por defecto ocupan toda la figura.
- Si se desean poner varios ejes en una figura se usa `subplot()` :

```
>> subplot(N,M,k);
```

- Organiza el espacio de la figura para que quepan NxM ejes de igual tamaño en N filas y M columnas, crea el eje k (desde 1 hasta NxM) y lo hace activo, por lo que el siguiente plot se hará sobre él:

```
>> x=[0:0.01:1]*pi;
```

```
>> subplot(211); plot(x,sin(x));
```

```
>> subplot(212); plot(x,cos(x));
```

- Si queremos ejes de distinta forma o tamaño dentro de una figura usaremos la función `axes()` :

```
>> p_eje=axes('Position', [x y ancho alto]);
```

Propiedades de los ejes

- Los ejes son los objetos gráficos que tienen más propiedades.
- Para cambiar muchas de sus propiedades existen funciones específicas:

`title('Evolución temperatura')` : pone un título global a los ejes

`xlabel('Tiempo (seg)'), ylabel('Temp (° C)')` : etiqueta ejes X Y

`grid on/off` habilita o deshabilita una malla sobre los ejes

`axis on/off` hace visibles/invisibles los ejes

`axis equal` mantiene la relación de aspecto en la pantalla.

`light` indica donde está la iluminación en unos ejes (superficie)

`view` marca la posición del observador en un gráfico 3D.

- Todos los comandos anteriores actúan sobre los ejes activos.
- Podemos obtener los ejes activos con `gca` (get current axes)

Comandos gráficos más comunes ($y=f(x)$)

- Dentro de los comandos que nos permiten hacer una grafica de una función $y=f(x)$, el más usado es `plot()`.
- Notar que MATLAB pinta colecciones de números, no funciones. Nosotros debemos generar previamente los x 's e y 's a pintar.

```
>> x=[0:100]*pi/100; y=sin(x); % Vectores a pintar (y frente a x)
>> figure; plot(x,y);
>> figure; plot(y);
```

- En el primer caso se pinta y frente a x (0 a π). En el segundo, al no tener información de x se pinta y frente al número de índice (0 a 100).
- Una opción es añadir una cadena texto con una letra indicando color elegido ('r','g','b') y un símbolo si deseamos línea no continua ('-' ':') o si queremos usar un marcador especial ('o' círculo, 's' cuadrado, etc.)
- Las propiedades por defecto del plot pueden cambiarse en el mismo comando, usando la notación ('Nombre propiedad', nuevo valor)

```
>> plot(x,y,'LineWidth',3,'Color','r')
```

Superposición de varios plots en unos ejes

- Dentro de unos ejes podemos tener varios gráficos (plots) al mismo tiempo (MATLAB usa automáticamente colores distintos).

```
>> plot(x,sin(x),x,cos(x),x,exp(-x)); % tres gráficas
```

- El comando `legend` permite identificar a los tres plots:

```
>> legend('sin(x)', 'cos(x)', 'exp(-x)');
```

- Si tratamos de pintar varios gráficos con llamadas repetidas a `plot` veremos que sólo el último permanece.

- La solución es usar el comando `hold on` que indica a MATLAB que los nuevos plots no deben machacar a los antiguos. Cuando terminemos de superponer gráficos haremos un `hold off`:

```
>> plot(x,sin(x), 'r'); hold on;
```

```
>> plot(x,cos(x), 'b');
```

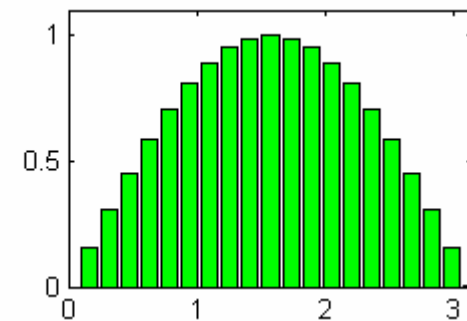
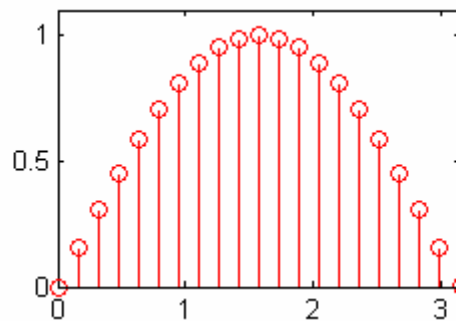
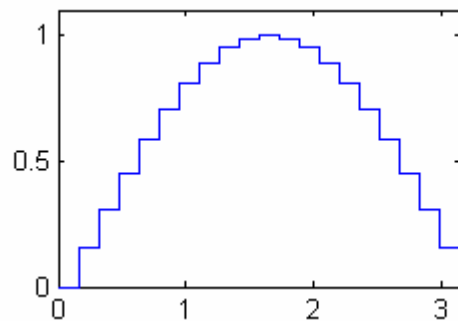
```
>> plot(x,exp(-x), 'g'); hold off
```

- En este caso debemos especificar nosotros colores distintos. Al ser plots separados MATLAB usaría siempre el color por defecto (azul).

Variantes de un plot

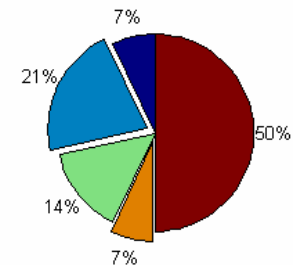
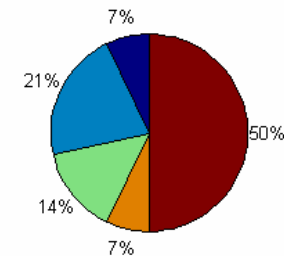
- Hay varios comandos que pintan gráficos similar a un plot (y frente a x), pero con una visualización distinta:

```
>> x=[0:20]*pi/20; y=sin(x); % Vectores a pintar (y frente a x)
>> subplot(131); stairs(x,y,'b');
>> subplot(132); stem(x,y,'r');
>> subplot(133,'g'); bar(x,y);
```



- Otra variante, un poco distinta es pie() que hace un gráfico circular. Su vector argumento debe ser positivo, al representar probabilidades:

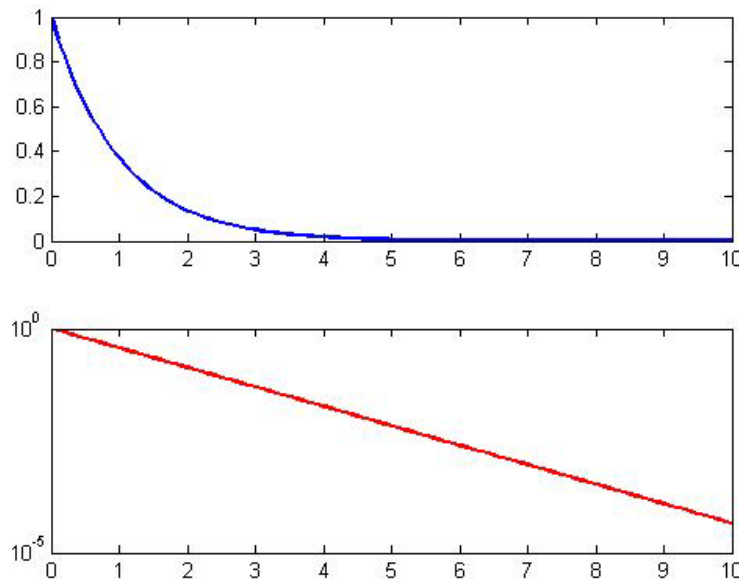
```
>> x=[1 3 2 1 7];
>> subplot(121); pie(x);
>> subplot(122); pie(x,[0 1 0 1 0])
```



Plots en ejes logarítmicos

- Si necesitamos usar ejes en escala logarítmica usaremos las funciones `semilogy` (escala log en Y), `semilogx` (escala log en X) o `loglog` (ambos ejes en escala log).
- Los parámetros que aceptan estas funciones son los mismos que un plot normal:

```
>> x = [0:0.1:10]; y = exp(-x);  
>> subplot(211); plot(x,y,'b','LineWidth',2);  
>> subplot(212); semilogy(x,y,'r','LineWidth',2);
```



Funciones gráficas ez (“easy”)

- Tradicionalmente para representar una función en un intervalo creamos un vector x barriendo el intervalo y calculamos el vector y aplicando la función. Luego hacemos un plot de y frente a x :

```
>> x=[0:100]*pi/100; y=sin(x);  
>> p=plot(x,y);
```

- En las últimas versiones de MATLAB se han añadido una serie de funciones gráficas que hacen más fácil (‘easy’ o ‘ez’) pintar gráficos.
- Por ejemplo la función `ezplot()` recibe una función a pintar (en forma de cadena texto) y un rango. La propia función se encarga de generar un vector x en el rango, evaluar la función dada para el vector y llamar al plot tradicional. Todo ello invisible para el usuario.

```
>> ezplot('sin(x)', [0 pi]); % Pinta sin(x) entre 0 y pi
```

- La desventaja es que no se tiene control de p.e. cuantos puntos se usan y sobre todo que se pierde la idea de que MATLAB sólo sabe manejar vectores y números.
- Es preferible entender primero la forma tradicional.